RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 1: Object-Oriented Development

Lecture Slides

Object-Oriented Development

- In general, ***development*** consists of three broad activities
  - These can be further subdivided but conceptually activities generally fall into one of these broader categories
- **Analysis (e.g. OOA)**
  - understanding and defining the problem
- **Design (e.g. OOD)**
  - A high level conceptual solution expressed using diagrams, pseudocode etc.
- **Programming (e.g. OOP)**
  - A detailed solution implemented in a programming language such as Java, C# or C++

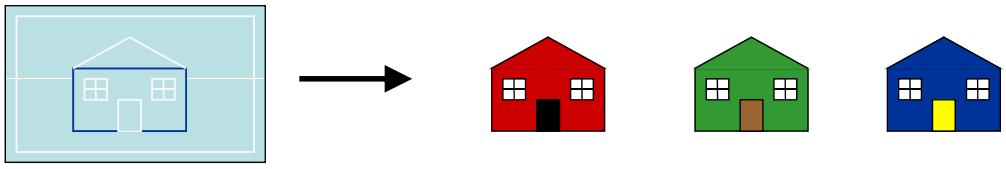
Background

- In *Programming 1* you have done mainly OOP
- In *Programming 2* you will do OOP and some introductory OOD
- OOA is covered in software engineering
- OOD is covered in more detail in software engineering, advanced electives and Programming 3

Revision: Classes and Objects

A class can be compared to a **blueprint** created by an architect when designing a house. It defines the important characteristics of the house, such as its walls, windows, electrical outlets, and so on.

Once we have the blueprint, several unique houses can be built, each with different addresses, furniture, colours, etc.



A 'blueprint' for a Java class contains attributes (variables) and methods. When an object is made out of the class, an **instance** of the class is said to be made (i.e. The class is *instantiated*). Unless explicitly specified (via static keyword), all of the properties of each instance are unique; for example, changing one instance's variable will not affect the other instances.

Revision: Object-Orientation in Java

Java is a purely object-oriented language: (almost) everything is an object.

- Java application programs are **built only of classes**
- Java provides a **very small core language** of primitive data types, operators and control structures; the remainder of the functionality is provided through a **very large class library** (the API.)
- Every primitive type in Java can be converted to an object, through the use of **wrapper classes**.

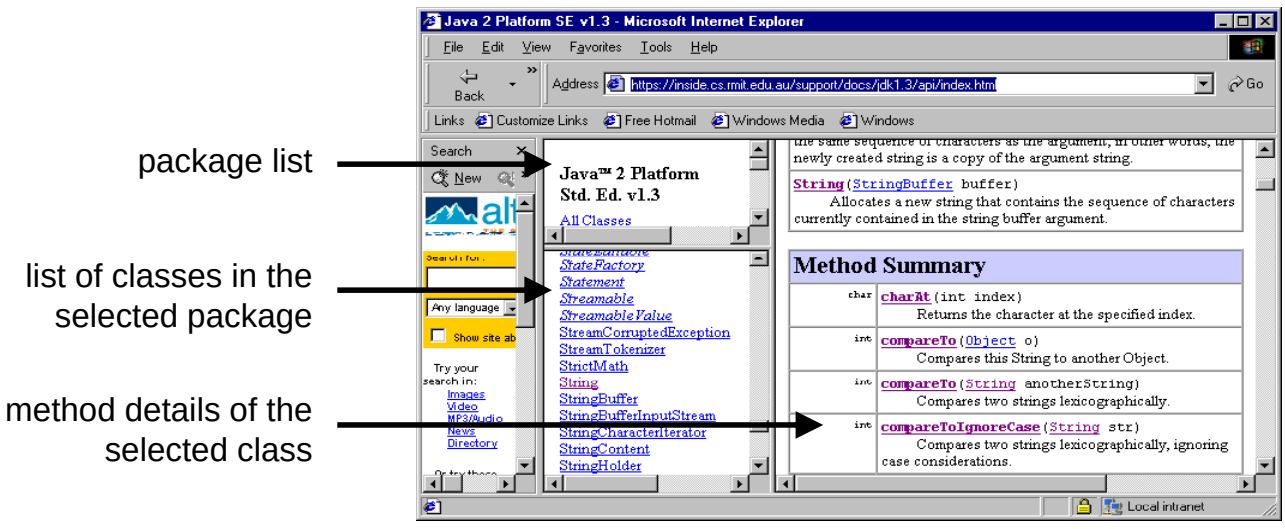
byte → Byte	float → Float	double → Double
Short → Short	long → Long	char → Character
int → Integer		boolean → Boolean

- Java supports all four key features of object programming; **encapsulation**, **inheritance**, **polymorphism** and **dynamic binding**.

Revision: Using the Java API Documentation

Java includes a large number of support classes with pre-defined functionality in its API (Application Programming Interface.)

The details of all these classes and their properties can be viewed easily using the Java Docs.



Example: a Bank System

Take the example of a **bank**.

- The **bank** stores data about its **customers**.
- A **customer** has a name, a customer number, a password and one or more accounts.
- An **account** is identified by an account number and has a balance.
- A customer can perform withdrawals, deposits and balance queries on their accounts.

How can we **discover classes** (and their relationships)?

How can we **discover attributes** (variables) **and methods for each class**?

Discovering Classes and Relationships (1)

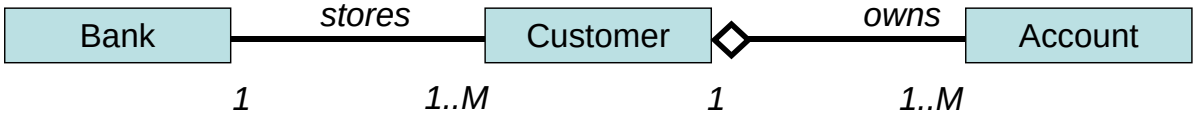
Classes are often either a **concrete entity** (a student, or a course), or an abstract concept (a shape, or form of transport.)

A good rule of thumb for discovering classes is to **look for nouns** in the problem specification.

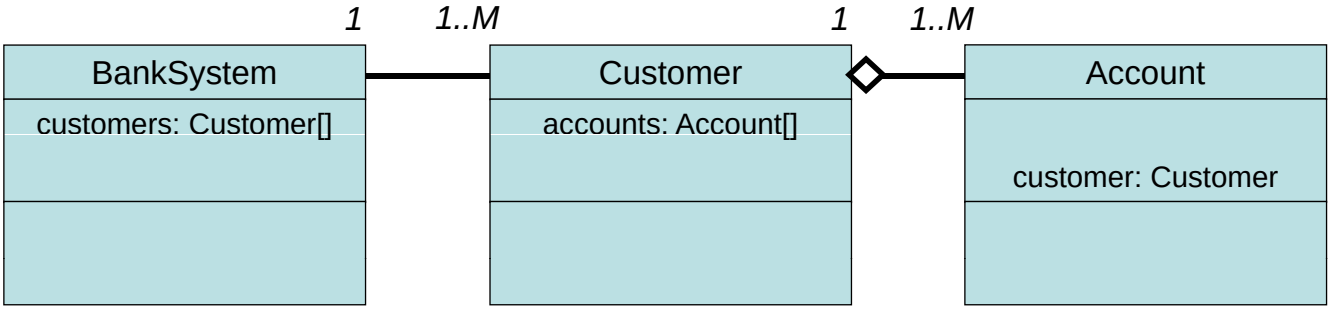
Class relationships can take different forms; **association**, **composition/aggregation** and **inheritance**.

In practice the difference between the first three and last form (inheritance) is the most important.

When determining cardinality (number of instances in a relationship e.g one to many or many to many) over-estimate rather than underestimate.



Representing Classes and Relationships (2)



Classes are represented by creating a new class type with the `class` keyword

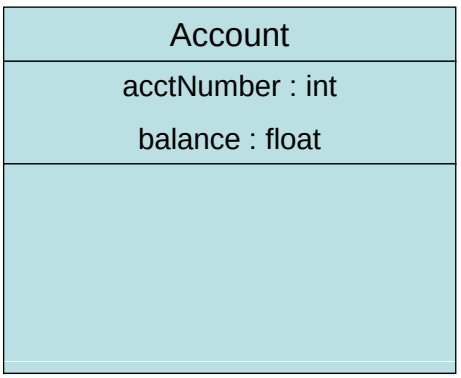
1..1 relationships are represented by a single attribute.

1..M relationships the M reference is held in the 1 class by some sort of collection (e.g. an array or vector). M .. M is basically a pair of 1 .. M collections.

Inheritance is represented using class inheritance in Java (`extends` keyword) and is potentially complex when using polymorphism, interfaces and abstract classes. However it is a powerful and sophisticated technique when done well!

This is covered in topics 2 and 3.

Discovering Attributes



To discover attributes, ask question such as:

"I am an account. What should I know? What do I need to remember?"

- my account number?
- my account balance?
- my owner?
- my bank?
- etc.

For each attribute, you need to decide on its name, data type, visibility (public, protected, private or default (package private)], also any modifiers i.e. should it be a constant (final) or variable, and having instance or class scope (static).

Discovering Methods and Relationships (1)

Account
acctNumber : int balance : float
withdraw(amount : float) : boolean deposit(amount : float) getAcctNumber() : integer getBalance() : float

Tip: look for verbs in the problem specification.

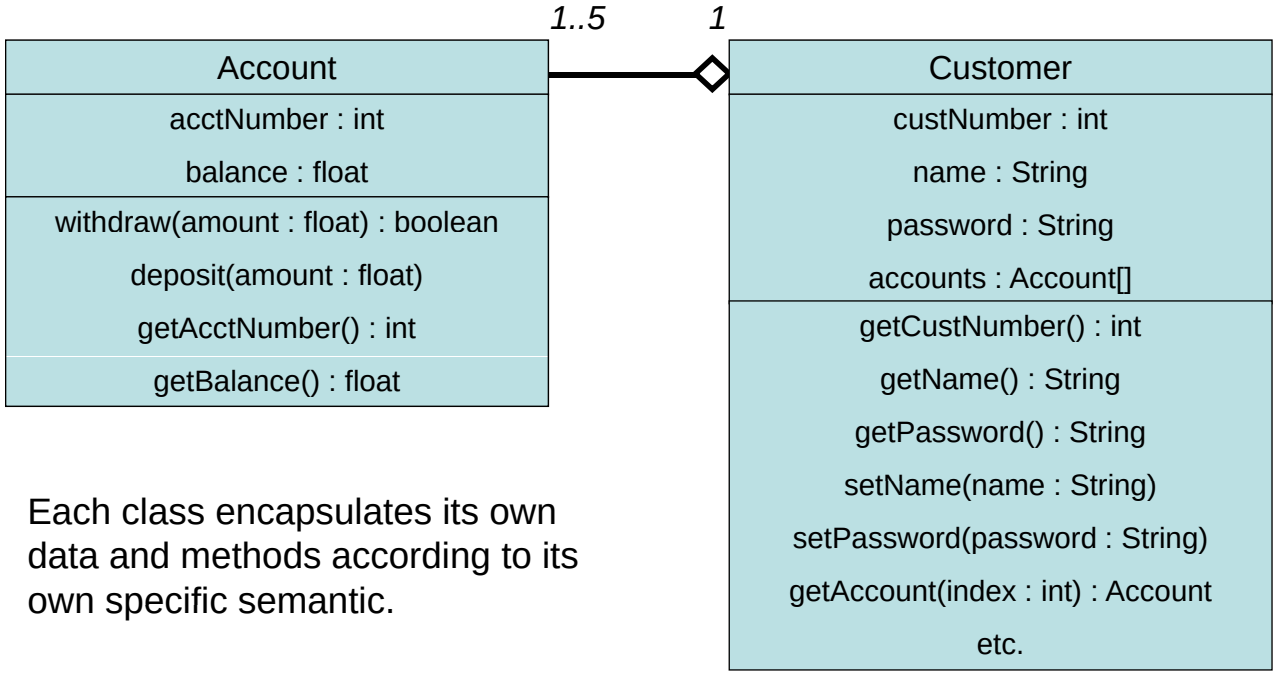
Ask question such as:

"I am an account. What services should I provide?" "What expertise should I have?" "What is my role?" "Who will be using me?" "Who do I interact with?"

- query my account number?
- query my account balance?
- change my account number?
- change my balance?
- deposit or withdraw money?
- etc.

Like attributes, for each method you also need to decide on its name, data types (for parameters and return value), visibility (public, protected, private or default (package)], and whether it should be final (not overridable) or static (called at the class level).

Discovering Methods and Relationships (2)



Each class encapsulates its own data and methods according to its own specific semantic.

The Account class: `Account.java` (1)

```
class Account {
  private int acctNumber = 0;
  private float balance = 0.0;

  public Account(int aNumber, float aBalance) {
    acctNumber = aNumber;
    balance = aBalance;
  }

  public int getAcctNumber() {
    return acctNumber;
  }
```

Variable declaration and initialisation

Class constructor. It has **no** return type, not even **`void`**.

Method definition

(in this case, an accessor method)

Note the use of **`private`** and **`public`**; encapsulation requires the support of **information hiding**, where internal implementation details must be kept hidden from the outside world. This is done with **visibility modifiers**.

The Account class: `Account.java` (2)

```
  public boolean withdraw(float amount){
    if (balance >= amount) {
      balance -= amount;
      return true;
    }
    else
      return false;
  }

  public void deposit(float amount) {
    balance += amount;
  }

  // other Account class methods would be added here
}
```

`withdraw()` returns a boolean to return the success of the operation. This is just a design decision. What else could be done?

The Constructor

The constructor is a **special method** that is invoked to create a new object. Its purpose is to **initialise the object** (including its instance variables) **to a known state**.

The constructor has the same name as the class. It is always invoked with the **new** keyword. It **never** has a return type.

```
Account a = new Account();
```

By default, every class is provided with a default constructor that takes no parameter. If a class provides its own constructor (that takes parameters), then the default constructor is **not** automatically provided.

In this case, the class ***must explicitly define*** the default constructor, if desired. Many classes have more than one constructor, each with a different parameter list (i.e. the constructors are overloaded.)

Accessors and Mutators

Data should be kept private to classes (in order to comply with information hiding.) In order to provide access to private data, we may need to provide accessor and mutator methods (also known as 'get' and 'set' methods, respectively.)

Accessors ('get' methods) retrieve the value of the attribute being accessed.

Mutators ('set' methods) modify the value of the attribute being accessed.

Accessors and mutators are usually provided for attributes, however in some cases it makes better sense to omit either (or both.) For example, should we provide a mutator for an account number?

Rule of thumb: provide accessors and mutators only when required, and when they make sense for a problem at hand.

Example Customer class: `Customer.java` (1)

```
class Customer {
  private String name = null;
  private int custNo = 0;
  private String password = null;

  private int numAccounts = 0;

  private Account[] accounts = new Account[MAX_ACCTS];

  private static final int MAX_ACCTS = 5;

  Customer(String name, int number){
    this.name = name;
    this.number = number;
  }

  public String getName(){
    return name;
  }
```

Keep track of how many accounts have been open so far

An array of `Account` objects

`final` makes it constant.

`static` makes it a class, rather than instance, attribute.

The Customer class: `Customer.java` (2)

```
  public int getCustNo(){
    return custNo;
  }

  public String getPassword(){
    return password;
  }

  public Account getAccount(int index){
    return accounts[index];
  }

  public Account[] getAccounts(){
    return accounts;
  }
```

Return only the account at a given index

Return the whole array of accounts

The Customer class: `Customer.java` (3)

```
    public boolean addAccount(Account a) {
      if (numAccounts < MAX_ACCOUNTS){
        accounts[numAccounts] = a;
        numAccounts++;
        return true;
      }
      else
        return false;
    }

    public void setPassword(String pw){
      password = pw;
    }

    // other methods go here
}
```

Account successfully added

Account array failed as it exceeded the array limit

Driver Program to Test the `Customer` Class

```
class Driver {
  public static void main(String[] args){
    Customer john = new Customer("John", 1234);
    Account jAcct = new Account(1, 300);
    john.addAccount(jAcct);

    Account returned = john.getAccount(1);
    System.out.println("John's account balance: " +
                            returned.getBalance());
  }
}
```

The driver program (main method) acts like a controller. All activities start and eventually end here.

Objects communicate by sending messages to each other.

Every method (and control structure within) should be tested * testing covered in more detail in software engineering course.

A Simple Bank System: `Bank.java`

```
class Bank {
  public static void main(String[] args){

    if (args.length < 2){
      System.err.println("Usage: java Bank user bal");
      System.exit(1);
    }

    Customer cust = new Customer(args[0], 1);
    float balance = Float.parseFloat(args[1]);

    cust.addAccount(new Account(1, balance));
    System.out.println("New account created!");

    Account ret = cust.getAccount(0);
    System.out.println("Balance: " + ret.getBalance());
  }
}
```

This holds the command line arguments. This program requires two.

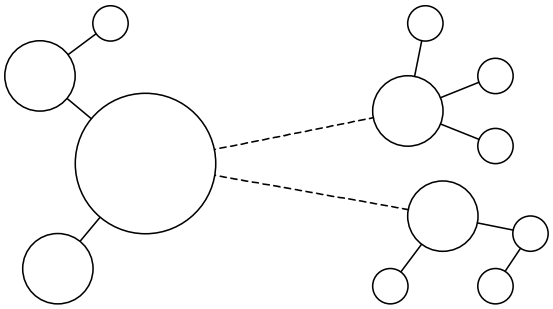
Parsing a `String` to a `float`.

Shorthand

Object-Oriented Design

As we have seen **object-oriented programming** is based on a **number of objects working together** to perform a function.

This kind of approach to developing code can give a number of benefits, such as easier **code maintenance** and enhanced **re-usability**.



a developers' object-oriented program

existing, re-usable software components

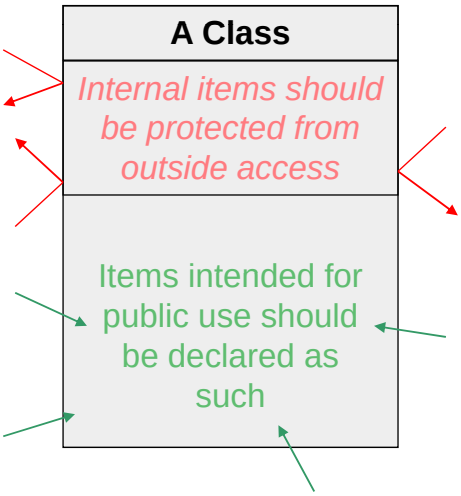
However, a **poorly designed** object-oriented program can be **just as, if not more difficult to use and maintain** than if it were developed using procedural or functional approaches.

Information Hiding

Now that all of Java's object capabilities have been detailed, we can review the fundamentals of **good OO design** with a **focus on implementation**.

Information hiding is a fundamental requirement to good OO design.

It means that outside access to the class' internals should be on a **"need to know basis"** – if it is not expected for a particular data item or feature to be publicly available, its visibility should be restricted.

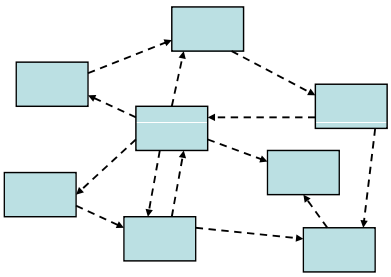


Coupling

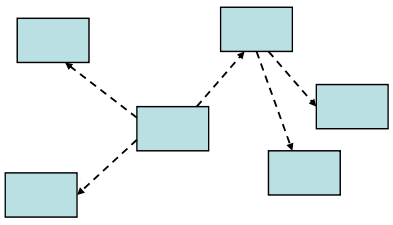
Classes should also be properly **encapsulated**. This refers to each class holding all the data/functionality it requires to fulfil its purpose, without being unduly dependent on others.

The dependence of some classes on others in this context is called **coupling**. This reliance should be minimised in your class designs wherever possible.

Excessive coupling between classes **can create problems** when items are modified – with so many dependencies, changing one will imply that many others may also need to be changed.



excessively high coupling



low coupling

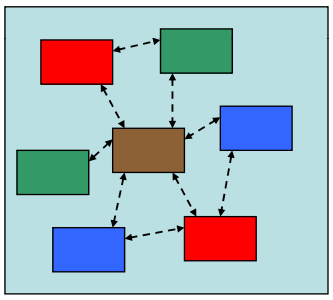
Cohesion

When a class is designed, each of its elements should have a **logical and well-defined role** in the class being able to fulfil its purpose. This **tight relation** of internal components is known as **cohesion**.

A **highly cohesive object** will **not** have items that are **only loosely related** to its role – the object will define **one** component well, and not attempt to be a “Jack of all trades”.

Cohesion is a **good** thing – it reflects a design that is **very clear and pure** to its intended purpose.

Coupling and Cohesion is a tradeoff – increasing cohesion (good) can increase coupling (bad).



A highly cohesive object will have elements that are closely inter-related.

Cohesion and Coupling Trade-off

- Cohesion and coupling are a trade-off
  - improving one may have a negative impact on the other
- e.g. Consider a program with only one class
  - it has low coupling (good) since there are no other classes to couple to
  - however cohesion will be low (bad) because the class will be responsible for everything
  - e.g. user interface, database management, unrelated domain logic (accounts, customers etc.)
  - => lots of unrelated methods = not cohesive
- Write cohesive classes first then try to minimise coupling

Cohesion and Coupling Trade-off (continued)

- Conversely imagine a system with a large number of very small and highly cohesion classes that only have one method
  - One method means only a single functionality so cohesion is high (good).
  - However this system will be highly coupled (bad) because all the small classes have to interact to get work done

'Good' Object Oriented Program Design

An object oriented program design should:

- have all internal information well protected from others
- have classes that have a clear and focused purpose (maximise cohesion)
- consist of classes that are highly independent (minimise coupling)

Adhering to these basic design principles should help design object-oriented programs that are **easily understood**, **easily maintainable**, and consist of **highly reusable** components.

A good program design can **save a lot of time** when it comes to implementation, by eliminating the need for "hacking and patching" code and class models to make them work.

OOD Notation – A Diagrammatic Approach

Class diagrams show the structure of a program at the class level, showing

methods and attributes of classes

relationship between classes,

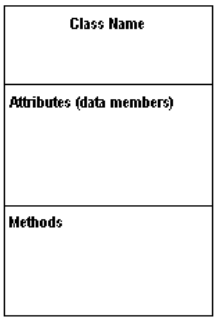
eg inheritance, aggregation

Can range from low (classes and relationships only) to high detail (e.g. full method signatures, visibility etc.)

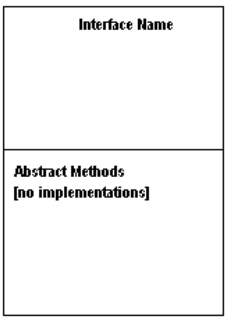
Class diagrams are part of a modelling language (UML) that you will encounter in later subjects in more detail.

OOD Notation – Class Diagram Approach (2)

A class representation in a class diagram

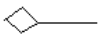


An interface representation in class diagram



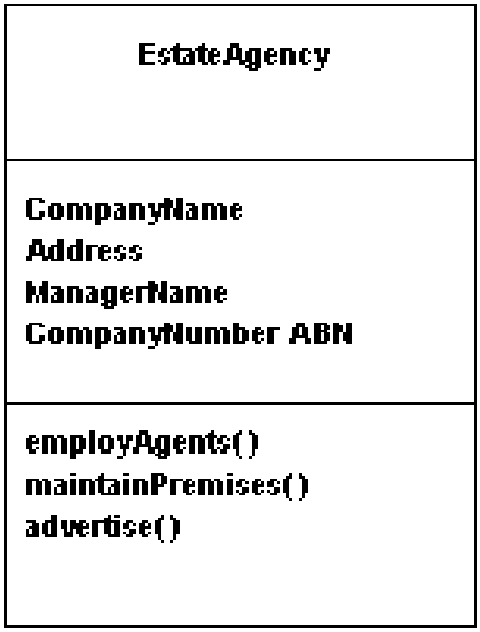
Inheritance
class-class
interface-interface

Association
a relationship
between classes



Aggregation / composition

OOD Example – Using diagrammatic approach



Exceptions and Error Handling

An exception defines an **unusual or erroneous situation** that can be handled by the programmer, for example an ArithmeticException (divide by zero), or an ArrayIndexOutOfBoundsException.

An error defines a generally **fatal machine-related problem** that cannot be handled by the programmer, so the program will terminate prematurely (crash.) For example, OutOfMemoryError.

Exception handling allows normal execution and exception execution flows to be separated. This **assists program design and implementation** as follows:

- **separation** allows normal execution to be made more efficient.
- some programs are required to **never** terminate abnormally.
- the **programmer can decide** where to handle an exception.
- it facilitates **debugging**.

Java Exceptions (1)

An exception is an object that can be returned from a method without the use of the return keyword. (An exception object is thrown, not returned.)

All objects that can be thrown are descendants of `java.lang.Throwable`. This class has two children, `java.lang.Error` and `java.lang.Exception`.

The class `java.lang.Exception` is very basic: the only data it holds is a single string, that can be used to hold an error message.

Classes that extend `Exception`, both library- and programmer-defined, usually maintain the same interface.

* `RuntimeExceptions` do not need to be caught or declared in method signature

Exceptions should be caught by the programmer – either:

- where the exception occurs, or
- elsewhere in the program (higher up in the method stack)
- If in doubt how to handle throw the exception rather than handle badly

Java Exceptions (2)

If an exception is not caught by the program, the virtual machine ultimately catches the exception, and terminates the program abnormally (it crashes.) When an exception is caught in this way, the error message, and the place where it occurred in the program, are output:

```
class Zero {
  public static void main(String[] args) {
      int a = 10;
      int b = 0;
      System.out.println( a/b );
   }
}
```

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```

The Exception Crash Message

```
java.lang.ArithmeticException: / by zero
   at Zero.main(Zero.java:7)
```

An exception crash message contains not only the message string (in this case, “/ by zero”), but also the method stack trace which specifies exactly where the error occurred, in the form method name (filename:line).

In general, a trace has more than one line.

Each line shows the method that was called to get to the method above, i.e. the trace represents the method calling hierarchy, and depicts the runtime stack.

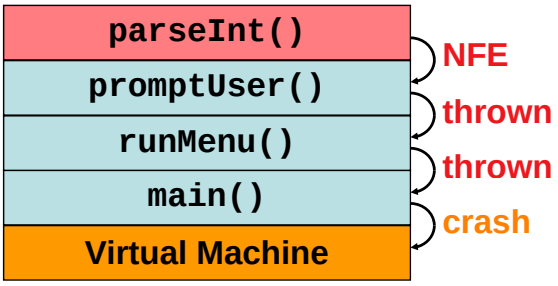
Exception Propagation

When an exception is thrown, it is moved along to the calling method in the method stack.

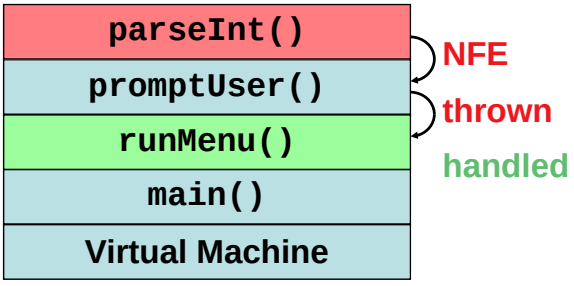
This process continues until it is either caught in a **`try..catch`** block, or it is thrown all the way to the Virtual Machine (which will cause the program to crash.)

The diagram on the right depicts how an exception can propagate through the method stack.

Without being caught:



Caught in `runMenu()`:



The `throws` Clause

The **`throws`** clause specifies that

(1) That exceptions are not going to be handled in the method where they are generated, and

(2) That they should be thrown to the calling method.

This has been used often in the past:

```
public static void main(String[] args) throws IOException {
   // code is here
}
```

All programs so far that included data input from the keyboard were defined as above.

The '**`throws IOException`**' tells Java that this method can throw an **`IOException`** to the calling method (that can be thrown from the **`readLine()`** method calls.)

The `try..catch` Block (1)

If, instead of crashing the program, we want to catch the exception and handle it, the exception-prone code can be isolated inside of a `try..catch` block, such as:

```
try {
  int a = Integer.parseInt("dog");
}
catch (NumberFormatException e) {
  System.err.println("Not an integer!");
}
```

If the call to **`parseInt()`** has a problem running, the **`NumberFormatException`** that it throws will make the **`catch`** block code run, instead of the exception being thrown.

The `try..catch` Block (2)

A **`try..catch`** block can have multiple **`catch`** statements.

However, due to the polymorphic behaviour of exception types, care must be taken to order them such that the most specific exception types are the first **`catch`** statements, and the more generic types caught last.

```
try {
   // source code here with
   // many possible exceptions
}
catch (NumberFormatException e){
   // handle this exception
}
catch (FileNotFoundException e){
   // handle this exception
}
catch (IOException e){
   // handle this exception
}
```


The `finally` Block

In some cases, there might be code that needs to be executed regardless of **whether or not** an operation completed successfully (for example, a file output stream that needs to be closed.)

A **`finally`** clause can be appended to the end of a **`try..catch`** block to allow this functionality.

```
try {
   // code
}
catch (exception-type) {
   // error handling
}
finally {
   // this code will always run
}
```

Creating an Exception Type (1)

To create a custom exception type, we can simply **extend** upon the appropriate subclass of the **`Exception`** hierarchy, and **inherit the properties** that make it usable as an exception.

For example, if our custom exception was to model a problem related to the process of inputting data, it would make logical sense to extend the **`IOException`** class.

All exceptions and errors inherit **basic error properties**, such as a **message** string and the **stack trace** information.

If those properties are all that we need (which is often the case), we simply extend the appropriate `Exception` class, and fill in the constructors.

Creating an Exception Type (2)

```
class RangeException extends Exception {

  RangeException() {
    super();
  }

  RangeException(String message) {
    super(message);
  }
}
```

Creating an Exception Type (2)

```
class RangeException extends Exception {

  RangeException() {
    super();
  }

  RangeException(String message) {
    super(message);
  }
}
```


Exceptions versus boolean return value

A boolean can be accidentally ignored, a checked exception must be caught and handled (and an unchecked exception still thrown at runtime).

A well named exception provides additional semantics/meaning about the exception whereas the boolean requires commenting/documentation to describe its purpose.

The boolean cannot distinguish between different error types, whereas you can use polymorphic catch blocks to catch multiple custom exceptions.

The main caution with exceptions is not to use them for generic message passing (i.e. throwing an exception object containing state rather than passing an object as a parameter).

In contrast, in cases where there is a single clear mode of failure that can be reasonably ignored then use a boolean.

Exception Test 1: Local (1)

```
public class ExceptionTest {
  private static void numberTest (int num) {
    boolean valid;

    try {
      if (num < 0 || num > 9)
        throw new RangeException("out of range, ");
      else
        valid = true;
    }
    catch (RangeException e) {
      System.out.print(e.getMessage());
      valid = false;
    }
```


Exception Test 1: Local (2)

```
    finally {
      System.out.print("always do this");
    }

    if (valid)
      System.out.print(", valid data");
  }
}
```

Testing Input:

When `main()` executes:

1. `numberTest(5);`
2. `numberTest(12);`
3. `numberTest(0);`
4. `numberTest(-1);`

Testing Output:

1. always do this, valid data
2. out of range, always do this
3. always do this, valid data
4. out of range, always do this

Exception Test 2: Hand Pass (1)

```
public class ExceptionTest2 {
  private static void numberTest (int num)
                        throws RangeException {
    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) {
    try {
      numberTest(5);
      numberTest(12);
      numberTest(0);
      numberTest(-1);
    }
```


Exception Test 2: Hand Pass (2)

```
    catch (RangeException e) {
      System.out.println(e.getMessage());
    }
    finally {
      System.out.println("Do this whatever happens");
    }
  }
}
```

Program Output:

A valid number was entered

Number out of range

Do this whatever happens

Exception Test 3: Do Nothing

```
public class ExceptionTest3 {

  private static void numberTest (int num)
                       throws RangeException {

    if (num < 0 || num > 9)
      throw new RangeException("Number out of Range");
    else
      System.out.println("A valid number was entered");
  }

  public static void main(String args[]) throws RangeException{

    numberTest(5);
    numberTest(12);
    numberTest(0);
    numberTest(-1);
  }
}
```

Program Output:

A valid number was entered

<crash; exception stack trace>

Exception Test 4: Combination (1)

```
public class ExceptionTest4 {
  private static int numberTest (int num, String s) {
    int convertedNum; int result = -1;
    try {
      if(num < 0 || num > 9)
          throw new RangeException("Number out of Range");
      convertedNum = Integer.parseInt(s);
      result = num / convertedNum;
    }
    catch( RangeException e) {
      System.out.println(e.getMessage());
    }
    catch( NumberFormatException e) {
      System.out.println(e.getMessage());
    }
    catch (ArithmeticException e) {
      System.out.println(e.getMessage());
    }
    return result;
  }
```

Here, a programmer defined exception is caught, and a library defined exception is passed on.

Where an exception is caught is a programmer's decision.
In general, handle an exception as near to the source as possible.

Exception Test 4: Combination (2)

```
  public static void main(String args[]) {
    int res;
    res = numberTest(5, "3");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(-1, "4");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(4, "sat");
    System.out.println(String.valueOf(res));
    System.out.println();
    res = numberTest(3, "0");
    System.out.println(String.valueOf(res));
    System.out.println();
  }
}
```


Exception Test 5: Variation on Test 4 (1)

Compare this with ExceptionTest 4.

```
public class ExceptionTest5 {
  private static int numberTest (int num, String s) {
    int convertedNum;
    int result = -1;
    try {
      if(num < 0 || num > 9)
          throw new RangeException("Out of Range");
      convertedNum = Integer.parseInt(s);
      result = num / convertedNum;
    }
```

Exception Test 5: Variation on Test 4 (2)

```
  catch(Exception e) {
    if (e instanceof RangeException)
      System.out.println(e.getMessage());
    else if (e instanceof NumberFormatException)
      System.out.println("Num. format exception");
    else if (e instanceof ArithmeticException)
      System.out.println("Arithmetic exception");
    else
      System.out.println("Unknown exception");
    }
  return result;
}
```


Java Exceptions and Repetition (1)

A common task is to repeatedly prompt the user for input until it is correct.

```
public class Adder {
  public static void main (String[] args) {
    int n1 = UserReader.getInt("Enter a number: ");
    int n2 = UserReader.getInt("Enter another number: ");

    System.out.println ("The sum is " + (num1+num2));
  }
}
```

The `UserReader` class is defined on the next slide.

Java Exceptions and Repetition (2)

```
class User_Reader {
  public static int getInt(String prompt) {
    BufferedReader stdin = new BufferedReader
                        (new InputStreamReader(System.in));
    int number = 0;
    boolean valid = false;

    while (! valid) {
      System.out.print (prompt);

      try {
        number = Integer.parseInt (stdin.readLine());
        valid = true;
      }
```


Java Exceptions and Repetition (3)

```
      catch (NumberFormatException exception) {
        System.out.println("Invalid input." +
                           "Try again.");
      }
      catch (IOException exception) {
        System.out.println ("Input problem." +
                            "Terminating.");
        System.exit(0);
      }

    }// end while loop

    return number;
  }
}
```